OptiInfra - Project Context & Design Document

AI-Assisted Development Guide for Windsurf

Version: 1.0
Date: October 16, 2025
Purpose: Context for AI-assisted code generation

🎯 PROJECT OVERVIEW

What We're Building

OptiInfra is a multi-agent AI platform that automatically optimizes LLM infrastructure to:

- **Cut costs by 50%** (spot instances, right-sizing, reserved instances)
- **Improve performance 3x** (latency optimization, KV cache tuning)
- **Ensure quality** (detect regressions before production)

The Problem We Solve

Companies running LLM infrastructure (vLLM, TGI, SGLang) waste **$50K-$500K per month** due to:

- Suboptimal configurations (40% waste)
- Idle GPU time (50% underutilization)
- Over-provisioning (fear of outages)
- Manual optimization (20 hrs/week engineering time)

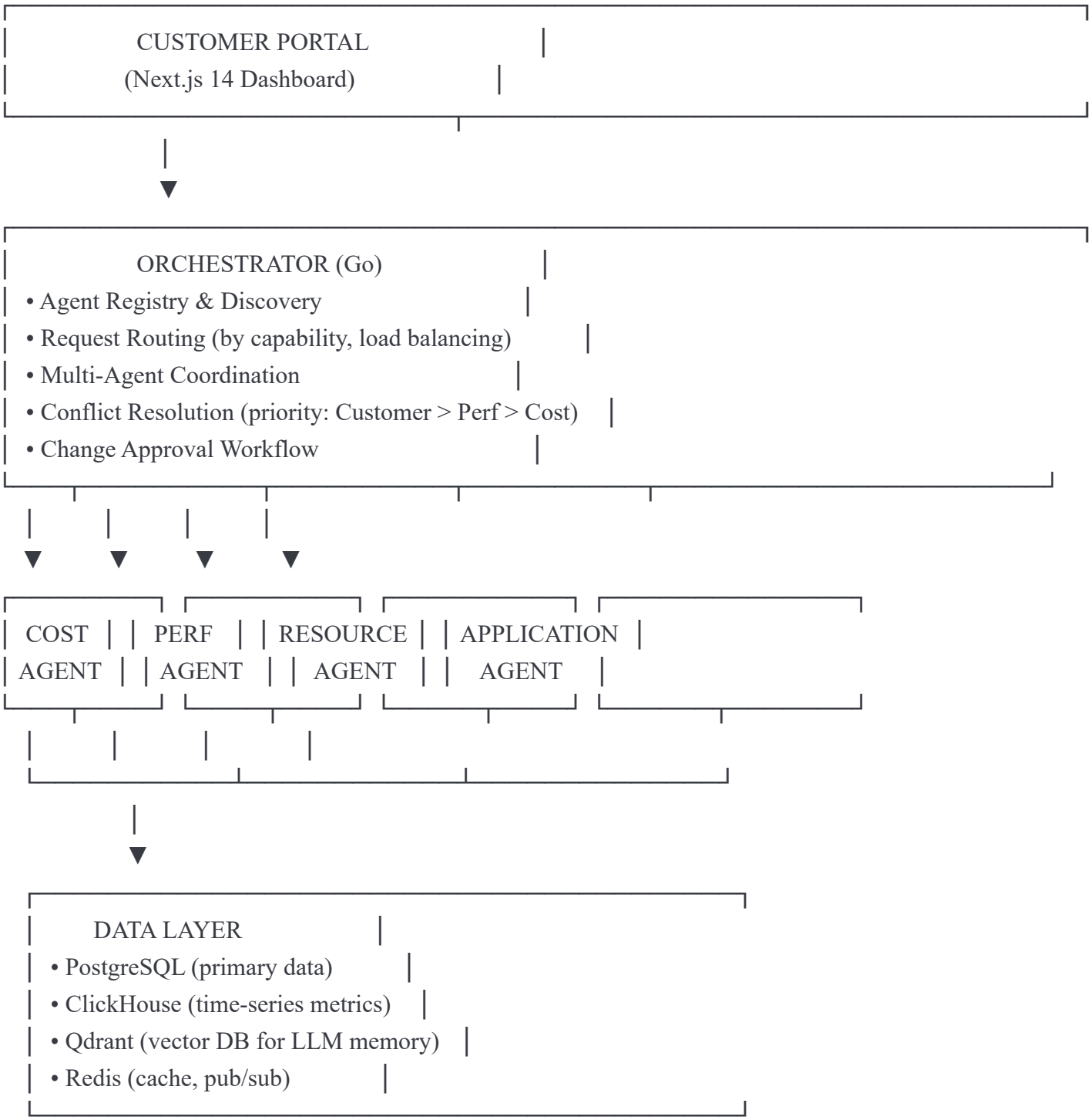
Our Solution

4 intelligent agents working together:

1. **Cost Agent** - Optimize cloud spending
2. **Performance Agent** - Improve latency/throughput
3. **Resource Agent** - Maximize GPU/CPU utilization
4. **Application Agent** - Monitor quality, prevent regressions

All coordinated by a **Go-based orchestrator**.

🧬 HIGH-LEVEL ARCHITECTURE



AGENT DETAILS

1. Cost Agent (Python/FastAPI + LangGraph)

Purpose: Reduce cloud spending

What It Does:

- Collects cost data from AWS/GCP/Azure (Cost Explorer, Billing API)
- Analyzes spending patterns (idle resources, over-provisioning)

- Generates recommendations (spot migration, right-sizing, RIs)
- Executes optimizations with approval
- Learns from outcomes (stores in Qdrant)

Key Workflows (LangGraph):

- Spot Instance Migration (30-40% savings)
- Reserved Instance Recommendations (40-60% savings)
- Instance Right-Sizing (20-30% savings)

Tech Stack:

- FastAPI (async API framework)
- LangGraph (workflow state machines)
- SQLAlchemy (database ORM)
- boto3/google-cloud/azure-sdk (cloud APIs)
- OpenAI/Anthropic (LLM for decisions)

---

2. Performance Agent (Python/FastAPI + LangGraph)

Purpose: Improve latency and throughput

What It Does:

- Collects performance metrics from vLLM/TGI/SGLang
- Identifies bottlenecks (high latency, low throughput, GPU underutilization)
- Generates optimizations (KV cache tuning, quantization, batch sizing)
- Tests in staging, gradual rollout (10% → 50% → 100%)
- Monitors for SLO violations, rollback if needed

Key Optimizations:

- KV Cache tuning (PagedAttention parameters)
- Quantization (FP16 → FP8 → INT8)
- Batch size optimization
- Model parallelism configuration

Tech Stack:

- FastAPI
- LangGraph
- Prometheus client (metric scraping)
- Integration with Application Agent (quality checks)

---

3. Resource Agent (Python/FastAPI + LangGraph)

Purpose: Maximize GPU/CPU/memory utilization

What It Does:

- Collects GPU metrics (nvidia-smi, utilization, memory, temperature)
- Collects CPU/memory metrics (psutil)
- Identifies underutilized resources
- Recommends scaling (up/down) or consolidation

- Integrates with KVOptkit for KV cache optimization

Key Workflows:

- Auto-scaling (predictive, based on trends)
- Resource consolidation (combine underutilized instances)
- KV cache optimization (memory allocation strategies)

Tech Stack:

- FastAPI
- LangGraph
- nvidia-smi (GPU metrics)
- psutil (CPU/memory metrics)
- KVOptkit integration

---

4. Application Agent (Python/FastAPI + LangGraph)

Purpose: Monitor quality, prevent regressions

What It Does:

- Monitors LLM output quality (relevance, coherence, factuality)
- Detects hallucinations and toxic content
- Establishes quality baselines
- Validates optimization changes (A/B testing)
- Auto-rollback if quality drops > 5%

Key Features:

- Quality metrics (per-request scoring)
- Regression detection (anomaly detection)
- Approval/rejection of other agents' changes
- A/B testing support

Tech Stack:

- FastAPI
- LangGraph
- Quality scoring models
- Statistical analysis (z-score, moving averages)

🧠 ORCHESTRATOR (Go)

Purpose: Coordinate all agents

Responsibilities:

1. **Agent Registry** - Agents register on startup, health monitoring
2. **Request Routing** - Route requests to appropriate agent(s)
3. **Conflict Resolution** - When agents conflict (e.g., Cost wants spot, Perf says too risky)
4. **Change Approval** - Workflow for approving optimizations
5. **Rollback Mechanism** - Revert changes if issues occur

6. **Event Publishing** - Notify agents of system events (Redis pub/sub)

Why Go?

- High performance (concurrent request handling)
- Low resource usage (compared to Python)
- Fast compilation
- Strong concurrency primitives (goroutines, channels)

Tech Stack:

- Gin (HTTP framework)
- gRPC (agent communication)
- Redis (pub/sub, registry backup)
- PostgreSQL (persistent state)

💾 DATA ARCHITECTURE

PostgreSQL (Primary Database)

Purpose: Store structured data

Tables:

- Customers (id, name, api_key, plan)
- Agents (id, type, status, last_heartbeat)
- Events (id, type, agent_id, data, timestamp)
- Cost metrics (customer_id, cloud, service, cost, timestamp)
- Performance metrics (service_id, latency_p95, throughput, timestamp)
- Resource metrics (instance_id, gpu_util, cpu_util, timestamp)
- Quality metrics (request_id, relevance_score, hallucination, timestamp)
- Recommendations (id, agent_id, type, savings, status)
- Optimizations (id, recommendation_id, executed_at, result)
- Approvals (id, optimization_id, approved_by, status)

ClickHouse (Time-Series Database)

Purpose: Store high-frequency metrics

Tables:

- cost_metrics_ts (1-minute granularity, 90-day retention)
- performance_metrics_ts (1-second granularity, 30-day retention)
- resource_metrics_ts (1-minute granularity, 90-day retention)
- quality_metrics_ts (per-request, 30-day retention)

Materialized Views:

- Hourly aggregations
- Daily aggregations

Qdrant (Vector Database)

Purpose: LLM memory and learning

Collections:

- cost_optimization_knowledge (past decisions → outcomes)
- performance_patterns (successful optimizations)
- customer_context (customer-specific learnings)

Usage:

- Store optimization attempts with embeddings
- Retrieve similar past decisions (vector search)
- Improve future recommendations based on history

Redis

Purpose: Caching and pub/sub

Use Cases:

- Agent registry (backup to PostgreSQL)
- API response caching (1-hour TTL)
- Rate limiting (per-customer, per-endpoint)
- Event pub/sub (notify agents of changes)

🔄 WORKFLOW EXAMPLE: Spot Migration

Step-by-step flow showing agent coordination:

1. COST AGENT: Analyze EC2 instances
   ├─ Query AWS Cost Explorer API
   ├─ Identify on-demand instances with stable workloads
   └─ Calculate potential savings (40%)

2. COST AGENT: Generate recommendation
   ├─ Use LLM to assess migration risk
   ├─ Create migration plan (blue-green deployment)
   └─ Estimate: Save $18K/month

3. ORCHESTRATOR: Coordinate agents
   ├─ Ask PERFORMANCE AGENT: "Is this safe?"
   │ └─ Performance Agent: "Yes, low-risk workload"
   ├─ Ask RESOURCE AGENT: "Will this work?"
   │ └─ Resource Agent: "Yes, sufficient spot capacity"
   ├─ Ask APPLICATION AGENT: "Establish quality baseline"
   │ └─ Application Agent: "Baseline recorded"
   └─ Route to customer for approval

4. CUSTOMER: Approve in portal
   └─ POST /api/recommendations/{id}/approve

5. COST AGENT: Execute migration
   ├─ Create spot instance
   ├─ Deploy workload (canary: 10%)
   ├─ Monitor for 5 minutes
   ├─ Scale up (50%)
   ├─ Monitor for 10 minutes
   ├─ Full migration (100%)
   └─ Terminate on-demand instance

6. APPLICATION AGENT: Monitor quality
   ├─ Compare quality metrics (before vs after)
   ├─ Quality drop: 0.5% (acceptable)
   └─ Approve migration

7. COST AGENT: Record outcome
   ├─ Store in Qdrant (success case)
   ├─ Update PostgreSQL (optimization complete)
   └─ Notify customer: "Saved $18K/month"

8. ORCHESTRATOR: Close workflow
   └─ Publish event: "spot_migration_complete"

🎨 CUSTOMER PORTAL (Next.js 14)

Purpose: Customer dashboard for viewing/approving optimizations

Pages:

1. **Overview** - All agents status, total savings, key metrics
2. **Cost Dashboard** - Cost trends, recommendations, approvals
3. **Performance Dashboard** - Latency/throughput trends, optimizations
4. **Resource Dashboard** - GPU/CPU utilization, scaling events
5. **Application Dashboard** - Quality metrics, regression alerts
6. **Recommendations** - All pending recommendations (approve/reject)
7. **Execution History** - Past optimizations, outcomes

Key Features:

- Real-time updates (WebSocket)
- Charts (Recharts library)
- Approval workflow (approve/reject recommendations)
- Execution monitoring (live progress)
- Dark mode support

Tech Stack:

- Next.js 14 (App Router)
- TypeScript
- TailwindCSS
- Recharts (charts)
- Axios (API client)
- WebSocket (real-time updates)

🔐 SECURITY & PRODUCTION

Authentication

- OAuth 2.0 (Google, GitHub)
- JWT tokens (access + refresh)
- Role-based access control (Admin, User, Viewer)

API Security

- Rate limiting (Redis-based, per-customer)
- API key authentication
- Input validation (Pydantic)
- SQL injection prevention
- XSS prevention
- CORS configuration

Deployment

- Kubernetes (all services)
- Horizontal Pod Autoscaler (auto-scaling)
- Health checks & readiness probes
- Resource limits (CPU, memory)
- Persistent volumes (databases)

CI/CD

- GitHub Actions
- Automated testing (unit, integration, E2E)
- Docker image builds
- Security scanning (Trivy)
- Staging → Production deployment

Monitoring

- Prometheus (metrics collection)
- Grafana (dashboards)
- Alerting (Slack notifications)

🛠️ TECHNOLOGY STACK SUMMARY

Component	Technology	Why?
Orchestrator	Go + Gin	Performance, concurrency
Agents	Python + FastAPI	Rapid development, ML libraries
Workflows	LangGraph	State machine management
Portal	Next.js 14 + TypeScript	Modern React, SSR
Primary DB	PostgreSQL	Reliability, ACID compliance
Metrics DB	ClickHouse	Time-series performance
Vector DB	Qdrant	LLM memory, similarity search
Cache	Redis	Speed, pub/sub
LLM	OpenAI/Anthropic	Decision-making, reasoning
Container	Docker + Docker Compose	Consistency, portability
Orchestration	Kubernetes	Production deployment
CI/CD	GitHub Actions	Automation
Monitoring	Prometheus + Grafana	Observability

📊 DEVELOPMENT PHASES (70 Prompts)

Phase 0: PILOT (Week 0) - 5 prompts

Goal: Validate AI-assisted development approach

- Bootstrap project structure
- Orchestrator skeleton (Go)
- Cost Agent skeleton (Python/FastAPI)
- LangGraph setup
- Simple spot migration workflow

Phase 1: Foundation (Week 1) - 15 prompts

Goal: Build infrastructure

- Database schemas (PostgreSQL, ClickHouse, Qdrant)
- Complete orchestrator (registry, routing, coordination)
- Shared utilities
- Mock cloud provider (testing)
- Monitoring (Prometheus/Grafana)

Phase 2: Cost Agent (Week 2-3) - 17 prompts

Goal: Complete cost optimization

- AWS/GCP/Azure collectors
- 3 workflows (spot, RI, right-sizing)
- Analysis engine, LLM integration
- Execution engine, learning loop
- API, tests, documentation

Phase 3: Performance Agent (Week 4-5) - 11 prompts

Goal: Performance optimization

- vLLM/TGI/SGLang collectors
- Bottleneck analysis, optimization engine
- Gradual rollout workflow
- API, tests, documentation

Phase 4: Resource Agent (Week 6-7) - 10 prompts

Goal: Resource optimization

- GPU/CPU/memory collectors
- Utilization analysis, KVOptkit integration
- Auto-scaling workflow
- API, tests, documentation

Phase 5: Application Agent (Week 8-9) - 9 prompts

Goal: Quality monitoring

- Quality metrics, regression detection
- Validation engine, A/B testing
- Quality workflow
- API, tests, documentation

Phase 6: Portal & Production (Week 10) - 8 prompts

Goal: Production-ready system

- Next.js portal (all dashboards)
- Authentication (OAuth/JWT)
- Kubernetes deployment

- CI/CD pipeline
- API security
- E2E system tests

🎯 KEY PRINCIPLES FOR CODE GENERATION

1. Production-Ready Code

- No placeholders or TODOs
- Complete error handling
- Comprehensive logging
- Input validation
- Security considerations

2. Testing

- 80%+ code coverage
- Unit tests for all functions
- Integration tests for workflows
- E2E tests for complete flows
- Mock fixtures for external services

3. Documentation

- Docstrings for all functions
- README for each component
- API documentation (OpenAPI)
- Architecture diagrams
- Troubleshooting guides

4. Consistency

- Follow language conventions (PEP 8 for Python, Go standards)
- Consistent naming (snake_case for Python, camelCase for Go)
- Consistent error handling patterns
- Consistent logging format

5. Performance

- Async/await where applicable
- Database connection pooling
- Caching (Redis)
- Query optimization
- Resource limits

6. Security

- Input sanitization
- SQL injection prevention
- XSS prevention
- Rate limiting
- Authentication/authorization

📝 CODE EXAMPLES & PATTERNS

FastAPI Agent Structure

python

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import logging

app = FastAPI(title="Cost Agent")
logger = logging.getLogger(__name__)

class AnalysisRequest(BaseModel):
    customer_id: str
    cloud: str  # aws, gcp, azure

@app.post("/analyze")
async def analyze_costs(request: AnalysisRequest):
    try:
        # Business logic
        result = await cost_analyzer.analyze(request)
        return result
    except Exception as e:
        logger.error(f"Analysis failed: {e}")
        raise HTTPException(status_code=500, detail=str(e))
```

LangGraph Workflow Pattern

python

```python
from langgraph.graph import StateGraph
from typing import TypedDict

class WorkflowState(TypedDict):
    customer_id: str
    instances: list
    recommendations: list
    approved: bool

def analyze_node(state: WorkflowState):
    # Analysis logic
    return {"recommendations": recommendations}

workflow = StateGraph(WorkflowState)
workflow.add_node("analyze", analyze_node)
workflow.add_node("recommend", recommend_node)
workflow.add_edge("analyze", "recommend")
```

Go Orchestrator Pattern

go

```go
package main

import (
    "github.com/gin-gonic/gin"
    "log"
)

type Orchestrator struct {
    registry *AgentRegistry
    router   *RequestRouter
}

func (o *Orchestrator) RouteRequest(c *gin.Context) {
    var req Request
    if err := c.ShouldBindJSON(&req); err != nil {
        c.JSON(400, gin.H{"error": err.Error()})
        return
    }

    agent := o.router.SelectAgent(req.Type)
    result := agent.Execute(req)
    c.JSON(200, result)
}
```

🎯 SUCCESS METRICS

Technical Metrics

- 80%+ test coverage (all agents)
- < 500ms API latency (P95)
- 99.9% uptime
- Zero critical security vulnerabilities

Business Metrics

- 40-50% cost savings demonstrated
- 2-3x performance improvement
- < 2% quality degradation (acceptable)
- 2-3 weeks to first savings

Development Metrics

- 70 prompts executed successfully
- < 10% manual code fixes

- 11 weeks total timeline (sequential)
- 4 decision gates passed

🚀 GETTING STARTED

This document provides context for AI-assisted development. The actual implementation follows 70 prompts in this order:

1. **Start with PILOT-01:** Bootstrap project structure
2. **Execute sequentially:** Respect dependencies
3. **Validate after each prompt:** Check success criteria
4. **Hit decision gates:** Evaluate at key milestones
5. **Track progress:** Use checklist

This context document will help you (Windsurf AI) generate high-quality, consistent, production-ready code throughout all 70 prompts.

Let's build OptiInfra! 🚀